NAT'L INST. OF STAND & TECH R.I.C.
A11103 520651

NIST
PUBLICATIONS

NISTIR 5737

A Method to Determine a Basis Set of Paths to Perform Program Testing

Joseph Poole

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards
and Technology
Gaithersburg, MD 20899

QC
100
.U56
NO.5737
1995

NIST

NISTIR 5737

A Method to Determine a Basis Set of Paths to Perform Program Testing

Joseph Poole

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards
and Technology
Gaithersburg, MD 20899

November 1995

U.S. DEPARTMENT OF COMMERCE
Ronald H. Brown, Secretary

TECHNOLOGY ADMINISTRATION
Mary L. Good, Under Secretary for Technology

NATIONAL INSTITUTE OF STANDARDS
AND TECHNOLOGY
Arati Prabhakar, Director

Abstract

A major problem in unit testing of programs is to determine which test cases to use. One technique that is in widespread use is to take the control flowgraph from each of the program functions and calculate a basis set of test paths. A basis set is a set of paths through the functions that are linearly independent and the paths in the basis set can be used to construct any path through the program flowgraph. Path construction is defined as adding or subtracting the number of times each edge is traversed. While this is not a total solution for test case generation, it does provide a good starting set of test cases. This paper gives an algorithm for taking a function's flowgraph and determining a basis set of paths. Proofs that the algorithm generates a set of paths that fulfill the above requirements are provided. A prototype tool, STest, is also discussed. Some general ideas for further improvement are also provided.

Keywords: basis testing; branch testing; complexity measure; cyclomatic complexity; program testing; STest; structured testing; testing

Contents

1 Introduction 4

2 The Algorithm and Proofs 7
2.1 Proof of Theorem 1 8
2.2 Proof of Theorem 2 9
2.3 Proof of Theorem 3 10
2.4 The Algorithm Generates the Marked Spanning Tree 12

3 STest 13

4 Conclusions 15

References 16

1 Introduction

One of the biggest problems in unit testing is how to determine test cases. The tester has to ensure that there are enough tests to do thorough testing, but not so many tests that all of the limited testing time is used up. There are several different techniques to pick test cases including using boundary cases, dataflow testing and branch testing[BEIZER90]. No single technique can supply sufficient test cases, so usually a combination of techniques are used. This paper address one of those techniques, basis set testing. A basis set is a set of linearly independent paths that can be used to construct any path through the program flow graph. In this paper, we examine a method that determines a set of paths for basis testing and give proofs that the algorithm fulfills the requirements of basis testing. In this section an overview of basis testing will be given. Section 2 presents the algorithm and proof of the algorithm. Section 3 describes Stest, a prototype testing tool which implements the algorithm. Section 4 presents conclusions.

Testing techniques can be divided into various categories. Black box testing ignores the internals of the software unit. Structured testing is, on the other hand, "Testing that takes into account the internal mechanism of a system or component"[IEEE610]. This can be further subdivided into different types including path testing, "Testing designed to execute all or selected paths through a computer program"[IEEE610] and branch testing, "Testing designed to execute each outcome of each decision point in a computer program"[IEEE610]. Basis testing, also known as Structured Testing[SP500-99], is a hybrid between these two techniques. The test paths in a basis set fulfill the requirements of branch testing and also test all of the paths that could be used to construct any arbitrary path through the graph.

Any function in a program[1] can be represented as a control flow graph. The nodes in this graph are program statements, while the directed edges are flow of control. Two nodes can be either unconnected, connected by an edge in either direction or connected by an edge in each direction. When tracing a path from the source to the sink, a backedge is a edge that leads back to a node that has already been visited. A flowgraph contains one source node and one sink. A source node is a node which has no incoming edges, while a sink node is a node with no outgoing edges. A program function may have more than one sink, but this graph can be converted into a graph with only one sink as described in section 2. Some languages also allow more than one source. This construct is very rare and not used in Structured Programming[LINGER79].

A basis set is a set of linearly independent test paths. A path can be associated with a vector, where each element in the vector is the number of times that an edge is traversed. For example, consider a graph with 4 edges: **a**, **b**, **c** and **d**. The path **ac** can be represented by the vector [1 0 1 0]. Paths are combined by adding or subtracting the paths' vector representations. Each path in the basis set can not be formed as a combination of other paths in the basis set. Also, any path through the control flow graph can be formed as a combination of paths in the basis set.

Figure 1 shows a simplified control flow graph. While a complete flow graph would not have two edges going to the same destination, this requirement has been relaxed to keep the number of paths to a manageable size for this example. A basis set for this graph is {**ac**, **ad**, **bc**}. The path **bd** can be constructed by the combination **bc** + **ad** - **ac** as shown in Table 1. The set {**ac**, **bd**} is not a basis set, because there is no way to construct the path **ad**. The set {**ac**, **ad**, **bd**} is also a basis set. Basis sets are not unique; thus a flowgraph can have more than one basis set.

---

[1]This is assuming that the language is a procedural language which is what most testing techniques assume.

McCabe's complexity measure[MCCABE76] is a software metric that attempts to evaluate how complex a function is. The number of paths in the basis set is equal to the complexity measure of that function. The value of the complexity measure is equal to the cyclomatic complexity of the flowgraph if all of the edges were undirected instead of directed. This can be calculated as equal to e - n + 2, where e is the number of edges and n is the number of nodes.

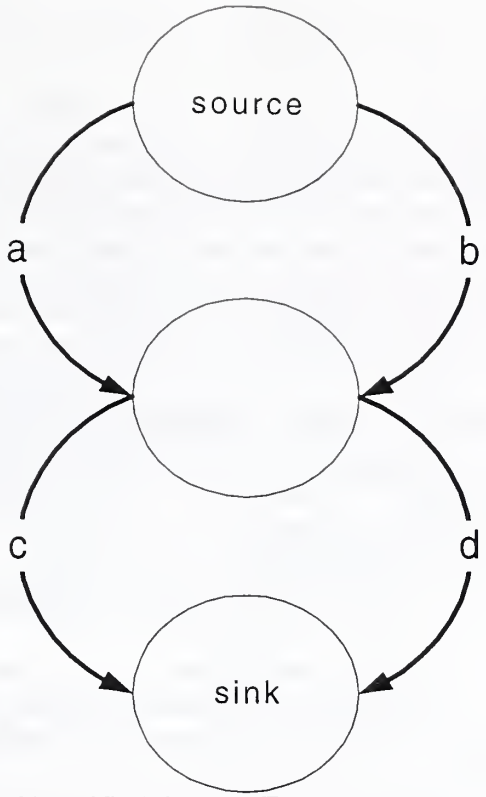

Figure 1. Simplified Control Flowgraph

Edge	**bd**	bc	bc + ad	**bc + ad - ac**
a	**0**	0	1	**0**
b	**1**	1	1	**1**
c	**0**	1	1	**0**
d	**1**	0	1	**1**

Table 1. Demonstration of Path Construction

2 The Algorithm and Proofs

The algorithm for our basis set method is a modified depth-first search algorithm. The search starts at the source node and recursively descends down all possible outgoing paths. If the node visited has never been visited before, a default outgoing edge is picked, then the current path is split into new paths that traverse each outgoing edge, going down the default edge first. The default edge is any edge which is not a back edge or which later causes a node to have two incoming edges. For example, in the test condition of a pre-test loop, the default edge would be the edge which exits from the loop. If the edge that traversed the body of the loop was chosen, then a back edge from the last node in the body to the test condition node would have to be traversed later. If the node visited is a sink (no exit edges), then a path in the basis set has been found. Otherwise, the path traverses the default edge. A pseudo-code outline of this method is as follows:

```
FindBasis( node)
        if this node is a sink
                print out this path as a solution
        else if this node has not been visited before
                mark the node as visited
                label a default edge
                FindBasis( destination of default edge)
                for all other outgoing edges, FindBasis( destination of edge)
        else
                FindBasis( destination of default edge).
```

The flowgraph from Figure 1 can be used to demonstrate this method. The algorithm starts at the source node. The default edge can be either edge, let us pick **a**. Edge **a** is traversed and the intermediate node visited. Edge **c** is picked as the default edge for the intermediate node. Edge **c** is traversed and the final node visited. There are no outgoing edges, so path **ac** is added to the basis set. The algorithm has traversed the default edge of the intermediate node, so now **d** is traversed. The destination is again the sink, so **ad** is added to the basis set. The intermediate node has had all of its edges traversed, therefore b from the source node is traversed. The destination of **b** is the intermediate node which has been visited before. Therefore the default edge **c** leading to the sink is traversed and the path **bc** is added to the basis set. The source has had all outgoing edges traversed, so the algorithm terminates. A program trace of this example would look as follows:

```
FindBasis( source)
        pick default ⇒ a
        FindBasis( destination(a)) ⇒FindBasis( intermediate)
                pick default ⇒ c
                FindBasis( destination(c)) ⇒FindBasis(sink)
                        print ac
                FindBasis( destination(d)) ⇒FindBasis(sink)
                        print ad
        FindBasis( destination(b)) ⇒FindBasis( intermediate)
                FindBasis( destination( default)) ⇒FindBasis(c) ⇒FindBasis( sink)
                        print bc
```

There are three assumptions that are required for this algorithm to work:

Assumption 1. There exists one source and one sink
Assumption 2. From the source, there exists a path to any node
Assumption 3. From any node, there exists a path to the sink

The algorithm will produce a basis set if there is more than one sink, but the number of paths determined is no longer equal to the complexity of the flowgraph. To convert a flowgraph with more than one sink to one with one sink, add edges to one sink from all of the other sinks in the graph. **Assumption 2** means that the function contains no unreachable code, while **Assumption 3** prohibits useless code.

For the paths generated by this algorithm to form a basis set, three theorems must be true:

Theorem 1. Each edge in the control graph will be traversed
Theorem 2. The set of paths in the basis set are linearly independent.
Theorem 3. Any path through the control graph can be formed as a combination of paths of the basis set.

Theorem 1 is required to show that the set of paths produced by the algorithm can be used for branch testing. **Theorem 2** and **Theorem 3** are needed to show that the paths are a basis set. Because of the complexity of the proofs for **Theorem 2** and **Theorem 3**, the proofs will be done in two steps. The first step is to generate paths using a flowgraph with certain edges marked and prove that **Theorem 2** and **Theorem 3** are true for that marked flowgraph. The second step will show that the algorithm produces the same set of paths as does the method using the marked flowgraph.

2.1 Proof of Theorem 1

To prove **Theorem 1**, we will first prove **Lemma 1.1** which states that all nodes will be visited, then show that all of the edges are traversed.

Lemma 1.1. All nodes will be visited.

This can be proven by induction on the length of the path between nodes.

Base Case: If a node is the destination of an edge from a visited node, then that node will be visited.

This follows because when the previous node is first visited, all outgoing edges will be traversed and their destination nodes will be visited.

Induction Step: Assume that any node that is n steps away from a visited node will be visited, prove that a node n+1 steps away will be visited.

When the parent of the n+1 node, which is n steps away, is visited, all of it's outgoing edges will be traversed. Since the n+1st node is a destination of one of these edges, it must be visited. This proves the inductive step. We start the algorithm at the source node and know from **Assumption 2** that a path can be traced from the source to any node, therefore **Lemma 1.1** is true.

On the first visit all edges out of a node will be traversed. Since **Lemma 1.1** shows that all the nodes will be visited, we know that all the outgoing edges of all nodes will be traversed. Therefore all of the edges in the control graph are traversed as **Theorem 1** states.

2.2 Proof of Theorem 2

Proving **Theorem 2** is more complex. First we will examine a flow graph and label some of the edges. We will then construct a set of paths from this labeled flowgraph and show that the paths are linear independent. In section 2.4, we will show that our algorithm generates this set of paths.

Label one outgoing edge of each node by performing a depth-first search from the sink and reversing the arcs. The marked edges form a spanning tree. A spanning tree is a graph where all of the nodes are connected and the removal of any edge causes the graph to become disconnected. See Figure 2 for a sample marked graph.

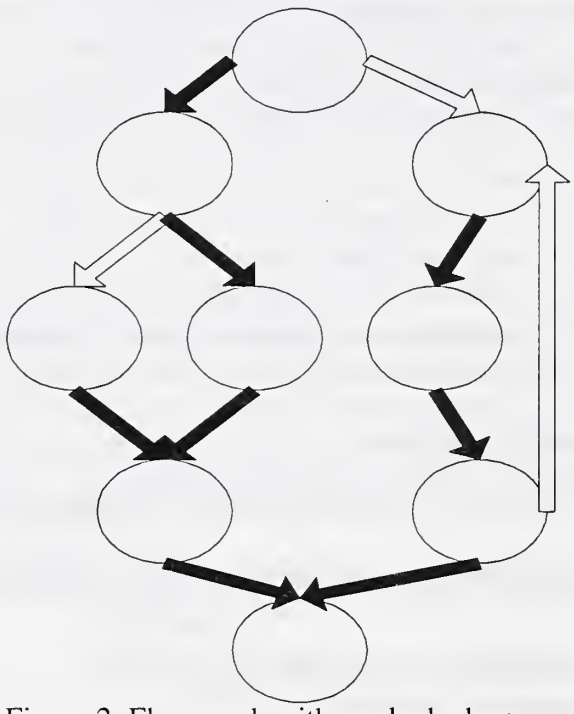

Figure 2. Flowgraph with marked edges

We will now prove three lemmas:

Lemma 2.1 The marked edges, ignoring arc direction, form a spanning tree.

Lemma 2.2 This is one and only one path of only marked edges from the source to the sink.

Lemma 2.3 There are e-n+1 chords (unmarked edges).

Lemma 2.1 and **Lemma 2.2** are true because the graph is generated by a depth first search. If e is the total number of edges and there are n-1 marked edges, there must be e-(n-1) unmarked edges, so **Lemma 2.3** is true. These edges are called chords.

We will now add paths to the set of basis paths. The goal is to associate each path with an unique chord. The one marked path through the graph is the first path to be placed in the basis set. Now add all additional paths from the source to the sink that include only one chord. There are some chords that are not on paths in the basis set. For each remaining chord **c**, it is impossible to trace a path from the source to **c** without including another chord **d**. Some paths will have more than one **d** chord. Chord **d** is called a required chord for **c**.

We know that there exists a separate path in the basis set that includes **d** but not **c**. The node at the head of **c** must have a marked outgoing edge since every node has one outgoing edge marked. This edge would have been chosen over **c** for traversal because this would be a path with only one chord.

All of the paths in the set are linearly independent, because each path is associated with an edge (the chord) that no other path contains. The only exception are the paths that have required chords. The path with both the required chord and the unique chord can not be eliminated from the basis set because it is the only path that contains the new chord. There is no combination that will include that chord. The path with only the required edge can not be eliminated because there are no paths that contain the new edge without the required chord. There is no way to create a linear combination that is equivalent to the path with only the required edge. This shows that **Theorem 2** is true.

2.3 Proof of Theorem 3

Theorem 3 states that any path through the flow graph can be generated as a linear combination of paths in the basis set. This will be shown by considering the arbitrary path **p** to be a combination of chords of the graph. To construct **p**, for each chord in **p**, add the basis set path associated with that chord the number of times the chord appears in **p**. The combination of paths will now have some extra traverses of marked edges. These can be removed by subtracting the path of only marked edges the required number of times.

For an example, use the flowgraph from Figure 1. Edges **a** and **c** are the marked edges. The basis set is { **ac**, **ad**, **bc**}. The path **bd** is created by adding the path containing the chord **b**, which is path **bc**, and the path **ad**, which contains the chord **d**. There is an extra occurrence of the edges **a** and **c**. These are removed by subtracting path **ac**, the path containing only marked edges.

An example using looping uses the flow graph in figure 3. The construction of path **adadbc** is presented in Table 2. The basis set of the flowgraph is { **ac**, **bc**, **adac**}. The chords are **b** and **d**. Constructing the path first uses the path associated with chord **b**, path **bc**, once because **b** only appears once in the target path. The other chord **d** appears twice so the basis path **adac** appears twice in the set. All of the chords are now in the set, but there are two extra occurrences of edges **a** and **c**. These are removed by subtracting the vector of the all marked-edge path **ac**.

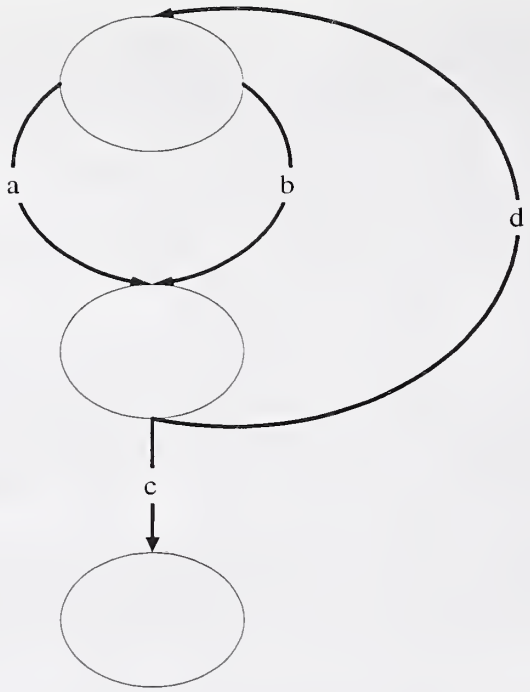

Figure 3. Simplified Flowgraph with Looping

Edge	adadbc	bc	bc + 2(adac)	bc + 2(adac) - 2(ac)
a	**2**	0	4	2
b	**1**	1	1	**1**
c	**1**	1	3	**1**
d	**2**	0	2	**2**

Table 2. Path Construction for Complex Example

2.4 The Algorithm Generates the Marked Spanning Tree

The final step is to show that the basis set algorithm generates the marked spanning tree used for the previous proofs. This can be shown to be true by examining the set of default edges generated by the algorithm. The default edges generated by the algorithm are the same as the marked edges of the spanning tree. Each node will have one outgoing edge marked. If the flows were reversed in direction, this would be equivalent to each node having one incoming edge marked. Since each node is visited and only n-1 edges are marked, the marked edges have to form a spanning tree. The default edge out of a node representing a test condition has to be carefully selected to ensure that all of the nodes are visited. For example, in a pre-test loop, the edge representing the exit case must be taken. If the edge representing the body of the loop was taken, the last node of the body will mark the edge going into the testing node. There would be no way to visit the node that is the destination of the exit edge because the node has already marked its one outgoing edge.

3 STest

STest is a prototype tool built to implement the basis set algorithm. It relies on another tool, **unravel**[NISTIR5691] to generate the program flow graphs. **unravel** is a static analysis tool that performs program slicing[WEISER84]. As part of it's processing, it generates an intermediate representation of the program including control and data flow. A LIF (language independent format) file is this intermediate format. LIF is described as follows in [NISTIR5691]:

> The *language independent format* represents the program as an annotated flow graph of nodes and edges. Nodes are generated to represent semantic or syntactic units of the program that correspond to statements or parts of statements. Edges are of two types, *control flow* and *requires*. A *control flow edge* between two nodes indicates the flow of control from one node to the other. The *requires* edges from a node indicate other nodes that should be included in any slice containing the node the edges are from. The *requires* edges are a general mechanism for specifying control dependence between nodes, pieces of required source code, or other slicing dependencies. The annotations specify location of corresponding source code, variables referenced or assigned and special statements such as **goto** and **return**.

The input of the *STest* program is a LIF file and the program source file. The LIF file contains the flow control graph information for all of the functions in that source code file. The LIF file also contains the mapping of the nodes to the source code file. The base name of the LIF file is assumed to be the same as the source code file. For example, file1.lif is the LIF file created from file1.c.

The program has two main components, the representation of the control flow graph and the parser function. The job of the parser is to take the LIF file and construct the control flowgraph. The parser function takes the representation as a input parameter and calls methods of the representation to create the control flowgraph.

The control flow graph is implemented as a FlowGraph object. This object contains data of each of the nodes in the source file, a list of which nodes are source nodes and the current path through the graph. Each node in the flow graph contains the number of incoming paths into the node, the number of outgoing paths into the node, any program text associated with the node, whether that node has been visited as part of the search and a linked list of all of the nodes which are adjacent to the node. There is a unique identification number, the NodeID, assigned to each of the nodes to distinguish them. The FlowGraph object has methods which perform the following operations:

- initialize a new flowgraph
- make a edge between two nodes
- add source code to a node
- make a node a source node
- traverse a node as part of a search
- traverse the entire set of graphs, starting at each source node.

The initialize step sets of the values for each node to defaults. Connecting two nodes adds a node to the starting node's linked list of adjacent nodes. It also increments the outgoing count of the start node and the incoming count of the ending node. Testing to see if two nodes are connected searches the linked list of the start node to see if the end node is present. The return value of the function is true if the end node is present, otherwise it is false. To add source code to a node requires the starting line and column of the text and the ending line and column. The program searches the source code file for the text and then copies it into the data for that node. Making a node a source node involves adding the NodeID number to a list of the source nodes.

Traversing a node is the heart of the program. The function checks to see if there are any outgoing paths from the node. If there are no outgoing edges, then this node is a sink and the path is printed out. If the node has not been visited, the function determines a default edge according to the algorithm. The function then marks the node as visited and then traverses all of the outgoing edges starting with the default edge. If the node was visited, then traverse is called recursively on the node that is the destination of the default edge. The program starts by traversing each start node in turn.

There are several extensions that can be made to improve STest. A major one is to add Myer's extension to the complexity measure[MYERS77]. The test condition for flow of control constructs can be a combination of conditions. This condition could be rewritten as two or more control constructs. For example, the test A and B for an if-clause can be rewritten as two nested if-clauses. The complexity for the first construct is 2, while the second construct has a complexity of 3. Both variations have the same effect. Myers proposed to extend the complexity measure by counting the boolean operations in the test condition. This would allow both constructs to have the same complexity value. In the first case, the construct has a complexity of 2 plus 1 for the "and" for a final complexity of 3. Myer's paper contains a fuller discussion.

4 Conclusions

This report presents a method for determining a basis set for testing. The basis set gives the unit tester a set of paths that can help in determining test cases for testing. The algorithm presented lends itself well to automatic testing. There are several limitations to the technique. The major limitation is that the method is unreliable on unstructured code. Useful test cases may be generated, but the number of paths in the basis set is no longer equal to the complexity measure of the flowgraph.

This algorithm provides a set of paths, but it does not provide data that can be used in the actual execution of the test cases. The data is impossible to provide for some functions. For example, in the following C function, there is no value of x that will execute the body in the first statement, but not execute the body in the second.

```
void f( int x) {
        if ( x < 5) y = 2;
        if ( x < 5) z = 1;
}
```

Basis set testing is not a complete testing solution, see [BEIZER90] and [EVANG84] for a discussion. Basis set testing does provide a good starting point. More test cases using other techniques can then be added for better testing coverage.

References

BEIZER90
B. Beizer, Software Testing Techniques, 2 Edition, Van Nostrand Reinhold, 1990.

EVANG84
M. Evangelist, "An Analysis of Control Flow Complexity", Eighth International Computer Software and Application Conference, pg. 388 - 396, 1984.

HETZEL84
W. Hetzel, The Complete Guide to Software Testing, QED Information Sciences, Inc., 1984.

IEEE610
ANSI/IEEE Std 610.12-1990, "Glossary of Software Engineering Terminology", The Institute of Electrical and Electronics Engineers, Inc., February, 1991.

LINGER79
R. Linger, H. Mills and B. Witt, Structured Programming: Theory and Practice, Addison-Wesley Publishing Company, 1979.

MCCABE76
T. McCabe, "A Complexity Measure", IEEE Transactions on Software Engineering, Vol SE-2, Number 4, December 1976, pg. 308-320.

MYERS77
G. Myers, "An Extension to the Cyclomatic Measure of Program Complexity", SIGPLAN Notices, October, 1977.

NISTIR5691
NIST IR 5691, J. Lyle, D. Wallace, J. Graham, K. Gallagher, J. Poole and D. Binkley, "Unravel: A CASE Tool to Assist Evaluation of High Integrity Software", volumes 1 and 2, Department of Commerce, August, 1995.

SP500-99
NBS Special Publication 500-99, T. McCabe, "Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Measure", December, 1982.

WEISER84
M. Weiser, "Program Slicing", IEEE Transactions on Software Engineering, 10:352-357, July 1984.